COMSC-210 Lab 2
Overloaded Operators, Templates, and Abstraction

GOOD PROGRAMMING PRACTICES *show / hide*

ABOUT THIS ASSIGNMENT
In this lab session, you will learn about overloaded operators, and templated classes. Overloaded operators are required, if you hope to use the C++ Standard Template Library (STL) for most of it's powerful features. The STL is based on the writing of templated classes, which make it possible to design and program the functionality of a class, and have it apply to data types that you could not foresee. By writing your own templated classes, you will gain a better understanding of how these work in the STL, and the ability to create your own special-purpose variations. The labs in this assignment are based on chapter 2 of our Childs textbook.

In ALL lab work involving test drivers, test ONLY the class that is the subject of the test. Do NOT test other classes that may also be used in the test, because those should already have been tested.

In ALL lab work involving templated classes, do NOT write class CPP files, as they do in our textbook. Do NOT #include CPP files *ever* in *anything*, even though the book does do so. Instead, write all function templates in the H file, below the class definition and before its `#endif`.

In ALL lab work involving classes (or structs), do NOT add to or modify the "public interface" as specified. That means do NOT add functions, or change function name spelling or casing (if specified). It's okay to add *private* functions as you wish, though, as they are not part of the public interface.

After you complete this lab assignment, post the required files to the COMSC server so that you receive the proper credit for this 50-point lab. Your posted files are due at midnight of the evening of the due date indicated in the course outline. Use the "Submit your FA2015 work" link on the class website to post your file for this lab, using the **lab2** folder provided.

LAB 2a: Writing Overloading Operators `[ Time.h, Time.cpp, and TimeDriver.cpp ]`
Purpose. The purpose of this lab is for you to learn how to create and apply the overloaded operators commonly used in C++ data structures.

Requirements. Modify the 3 lab 1b files: **`Time.h`**, **`Time.cpp`**, and **`TimeDriver.cpp`** by adding overloaded operators. Add these overloaded operators:

1. less-than, comparing two Time objects based on their time value in total seconds
2. equals, comparing their total time values -- *not*< their individual hours, minutes, and seconds. Note that a time value of 1 hour equals a time value of 3600 seconds.

You may write these overloaded operator functions as getter member functions or stand-alone, as you wish. Put any stand-alone prototypes in Time.h, AFTER the closing curly brace and semicolon of the class definition, and ABOVE the closing #endif. Put any stand-alone function definitions in Time.cpp WITHOUT a `Time::` scope resolution.

Modify the driver CPP so that it fully tests your overloaded operators.

Program I/O. Input: All hard-coded in the driver CPP only. Output: From the driver CPP only, console (`cout`) output only.

LAB 2b: Writing Templates And Overloading Operators `[ Road.h and RoadDriver.cpp ]`
Purpose. The purpose of this lab is for you to how to program with overloaded operators and templates.

Requirements. Modify your 3 files from lab 1a: **`Road.h`**, **`Road.cpp`**, and **`RoadDriver.cpp`** so that it uses a templated data type for the length and width, and for the return type of the asphalt function. You'll have to combine the Road class H and CPP files, because this new version is to be "templated". Also add these overloaded operators:

1. less-than, comparing two Road objects based on their length
2. equals, comparing their length *and* width

You may write these overloaded operator functions as getter member functions or stand-alone, as you wish.

Modify the driver CPP so that it fully tests your overloaded operators. Also include tests for two different data types (e.g., `int` and `double`) to confirm that your template specification works right.

NOTE: For this and all future lab work in this course, it's NOT okay to write a separate templated class CPP file to contain the function templates, as modeled in our textbook. The textbook's way of writing templated classes is to use separate H and CPP files, but the way shown in lecture is to write only an H file. Both ways work and are equally valid in real-life practice, but this is in the specifications for COMSC-210. This applies to templated classes *only* -- do NOT put non-templated function definitions in H files.

Submit the two files (1 CPP and 1 H) to the class website for credit.

Program I/O. Input: All hard-coded in the driver CPP only. Output: From the driver CPP only, console (`cout`) output only.

LAB 2c: Using Templated Classes `[ Calculator2.cpp ]`
Purpose. The purpose of this lab is for you to learn how to apply the STL `stack` template to a practical problem solution.

Requirements. Write 1 file: **`Calculator2.cpp`**, to solve exercise #14 on page 47-49 of the Childs textbook. But instead of using the Stack and Array classes referred to in the exercise, use an STL `stack` instead. You choose the data type for your stack -- there are more ways than one to do this right. Also, consider only the operations plus, minus, multiply, and divide, using floating point numbers.

Use the console for input, one token at a time, separated by pressing ENTER after each. Let an uppercase or lowercase Q terminate the program. Refer to the "how to's" at the bottom of the Lecture Topic 1 outline for dealing with strings. Note that 0 (zero) is a valid input -- if a user enters a 0, then a 0 should appear on the stack. In fact, if a user enters anything that resolves to zero, other than q or Q, consider it to be a zero.

DO NOT consider parentheses for forcing the order of operation. DO NOT worry about division by zero or any other numeric validation. DO avoid popping from the stack when the stack is empty -- for example, if

the user enters a plus operator, and there are fewer than two values in the stack, ignore the user's request.

The solution will be discussed in class during lecture. Submit the CPP file to the class website for credit.

Program I/O. Input: From the console (`cin`), one entry per line. Remember to NOT read numerics directly from `cin` -- read them as strings and convert. Do NOT type multiple entries on the same line of input. Include the current stack contents in the input prompt. Output: To the console (`cout`). Formatting of output precision is not required or recommended, but if you choose to do so, remember to NOT use the manipulator `fixed` because it is not compatible with all compilers.

Example (computer prompts in bold). Your program should match these RESULTS, formatted as you choose.

```
Example: five plus six
Enter:  5 [ENTER]
Enter:  5 6 [ENTER]
Enter:  6 5 + [ENTER]
Enter:  11.000000 Q [ENTER]

Example: ten minus one
Enter:  10 [ENTER]
Enter:  10 1 [ENTER]
Enter:  1 10 - [ENTER]
Enter:  9.000000 Q [ENTER]

Example: one divided by two
Enter:  1 [ENTER]
Enter:  1 2 [ENTER]
Enter:  2 1 / [ENTER]
Enter:  0.500000 Q [ENTER]
```

```
Example: Pi times the radius squared
Enter:  3.14159 [ENTER]
Enter:  3.14159 18 [ENTER]
Enter:  18 3.14159 18 [ENTER]
Enter:  18 18 3.14159 * [ENTER]
Enter:  324.000000 3.14159 * [ENTER]
Enter:  1017.875160 Q [ENTER]

Example: (1+2)/(3+4)
Enter:  1 [ENTER]
Enter:  1 2 [ENTER]
Enter:  2 1 + [ENTER]
Enter:  3.000000 3 [ENTER]
Enter:  3 3.000000 4 [ENTER]
Enter:  4 3 3.000000 + [ENTER]
Enter:  7 3.000000 / [ENTER]
Enter:  0.428571 Q [ENTER]
```

How to pause a console program: *show / hide*

GRADING RUBRIC *show/hide*
